# Module-2

## Python String:

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.
In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in Python to create a string.

**Syntax:**

```
str = "Hi Python !"
```

Here, if we check the type of the variable **str** using a Python script
**print**(type(str)), then it will **print** a string (str).

**Creating String in Python:**

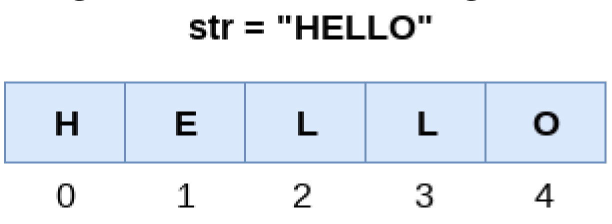
We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)

#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
str3 = '''Triple quotes are generally used for
    represent the multiline or
    docstring'''
print(str3)
```

**Output:**

```
Hello Python
Hello Python
Triple quotes are generally used for
    represent the multiline or
    docstring
```

**Strings indexing:**
Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

Consider the following example:

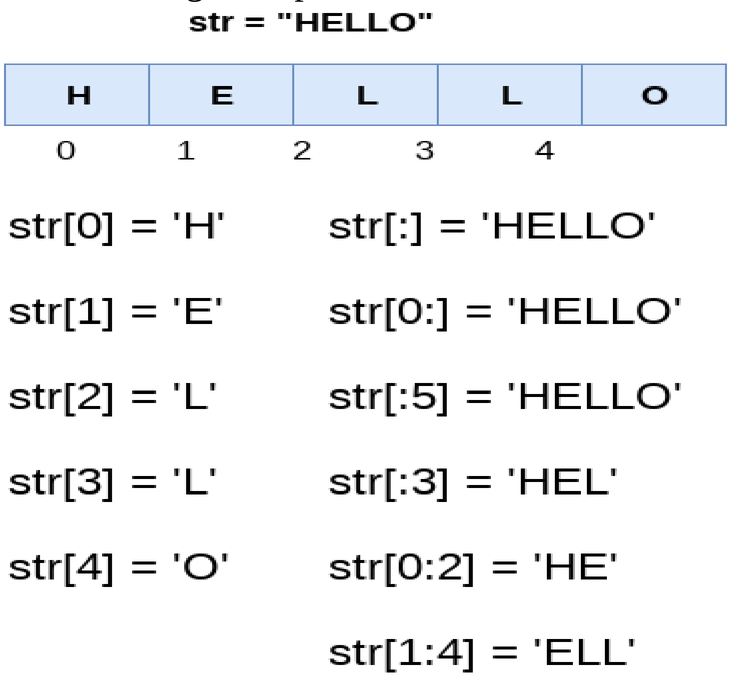
```
str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
# It returns the IndexError because 6th index doesn't exist
print(str[6])
```

**Output:**

```
H
E
L
L
O
IndexError: string index out of range
```

**Slicing:**
You can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.  As shown in Python, the slice operator **[]** is used to access the individual characters of the string. However, we can

use the **: (colon)** operator in Python to access the substring from the given string. Consider the following example.



```
str[0] = 'H'      str[:] = 'HELLO'
str[1] = 'E'      str[0:] = 'HELLO'
str[2] = 'L'      str[:5] = 'HELLO'
str[3] = 'L'      str[:3] = 'HEL'
str[4] = 'O'      str[0:2] = 'HE'
                  str[1:4] = 'ELL'
```

Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.

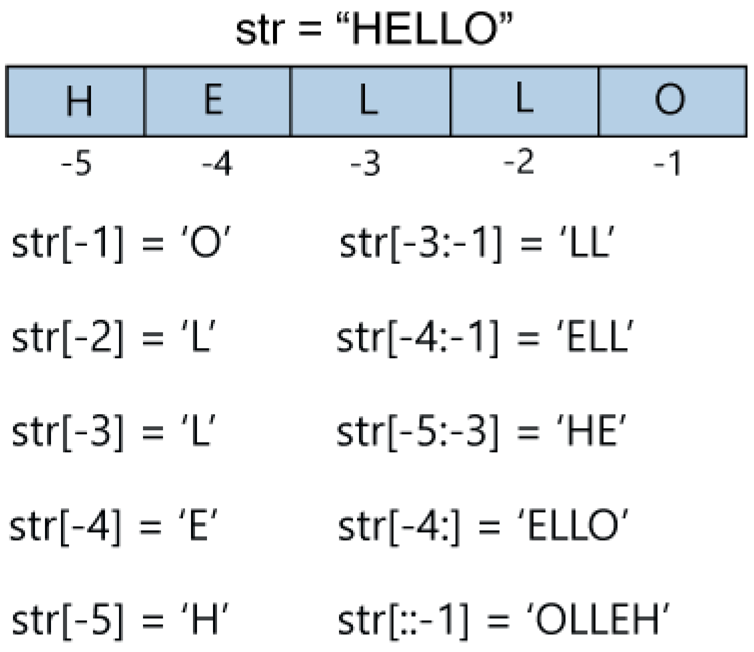
Consider the following example:

```
# Given String
str = "JAVATPOINT"
# Start 0th index to end
print(str[0:])
# Starts 1th index to 4th index
print(str[1:5])
# Starts 2nd index to 3rd index
print(str[2:4])
# Starts 0th to 2nd index
print(str[:3])
#Starts 4th to 6th index
print(str[4:7])
```

**Output:**

```
JAVATPOINT
AVAT
VA
JAV
TPO
```

We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on. Consider the following image.



Consider the following example

```
str = 'JAVATPOINT'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string
print(str[::-1])
print(str[-12])
```

**Output:**

```
T
I
NT
OIN
ATPOI
TNIOPTAVAJ
IndexError: string index out of range
```

**Reassigning Strings:**

Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. Strings are immutable in Python.

Consider the following example.

**Example 1:**

```
str = "HELLO"
str[0] = "h"
print(str)
```

**Output:**

```
Traceback (most recent call last):
  File "12.py", line 2, in <module>
    str[0] = "h";
TypeError: 'str' object does not support item assignment
```

However, in example 1, the string **str** can be assigned completely to a new content as specified in the following example.

**Example 2:**

```
str = "HELLO"
print(str)
str = "hello"
print(str)
```

**Output:**

```
HELLO
hello
```

**Deleting the String:**

As we know that strings are immutable. We cannot delete or remove the characters from the string.  But we can delete the entire string using the **del** keyword.

```
str = "JAVATPOINT"
del str[1]
```

**Output:**

```
TypeError: 'str' object doesn't support item deletion
```

Now we are deleting entire string.

```
str1 = "JAVATPOINT"
del str1
print(str1)
```

**Output:**

```
NameError: name 'str1' is not defined
```

## String Operators:

| Operator | Description |
|---|---|
| + | It is known as concatenation operator used to join the strings given either side of the operator. |
| * | It is known as repetition operator. It concatenates the multiple copies of the same string. |
| [] | It is known as slice operator. It is used to access the sub-strings of a particular string. |
| [:] | It is known as range slice operator. It is used to access the characters from the specified range. |
| in | It is known as membership operator. It returns if a particular sub-string is present in the specified string. |
| not in | It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string. |

**Example:**
Consider the following example to understand the real use of Python operators.

```
str = "Hello"
str1 = " world"
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
```

## Python String functions:
Python provides various in-built functions that are used for string handling.

| Method | Description |
|---|---|
| capitalize() | It capitalizes the first character of the String. This function is deprecated in python3 |
| count(string,begin,end) | It counts the number of occurrences of a substring in a String between begin and end index. |

| | |
|---|---|
| find(substring,beginIndex, endIndex) | It returns the index value of the string where substring is found between begin index and end index. |
| isalnum() | It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false. |
| isalpha() | It returns true if all the characters are alphabets and there is at least one character, otherwise False. |
| isdecimal() | It returns true if all the characters of the string are decimals. |
| isdigit() | It returns true if all the characters are digits and there is at least one character, otherwise False. |
| isnumeric()<br>**isdecimal()** **isdigit()** **isnumeric()**<br>**Example** of string with decimal characters: "12345" "12" "98201"<br>**Example** of string with digits: "12345" "123$^{3}$" "$^{3}$"<br>**Example** of string with numerics: "12345" "½¼" "½" "12345½" | It returns true if the string contains only numeric characters. |
| islower() | It returns true if the characters of a string are in lower case, otherwise false. |
| isupper() | It returns true if characters of a string are in Upper case, otherwise False. |
| lower() | It converts all the characters of a string to Lower case. |
| len(string) | It returns the length of a string. |
| swapcase() | It inverts case of all characters in a string. |
| title() | It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut. |
| upper() | It converts all the characters of a string to Upper Case. |

## Python List:

A list in Python is used to store the sequence of various types of data. Python lists are **mutable** type, its mean **we can modify its element** after it created. However, Python consists of **six data-types** that are capable to store the sequences, but the most common and reliable type is the **list**.
A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma "**,**" and enclosed with the square brackets [].

A list can be define as below

```
1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]
```

If we try to print the type of L1 and L2 using type() function then it will come out to be a list.

```
1. print(type(L1))
2. print(type(L2))
```

**Output:**

```
<class 'list'>
<class 'list'>
```

**Characteristics of Lists:**
The list has the following characteristics:

- The lists are ordered.
- The element of the list can access by index.
- The lists are the mutable type.
- The list elements are mutable types.
- A list can store the number of various elements.
- Since lists are indexed, lists can have items with the same value.

Let's check the first statement that lists are the ordered.

```
a = [1,2,"Peter",4.50,"Ricky",5,6]
b = [1,2,5,"Peter",4.50,"Ricky",6]
a ==b
```

**Output:**

```
False
```

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.
Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

```
a = [1, 2,"Peter", 4.50,"Ricky",5, 6]
b = [1, 2,"Peter", 4.50,"Ricky",5, 6]
a == b
```

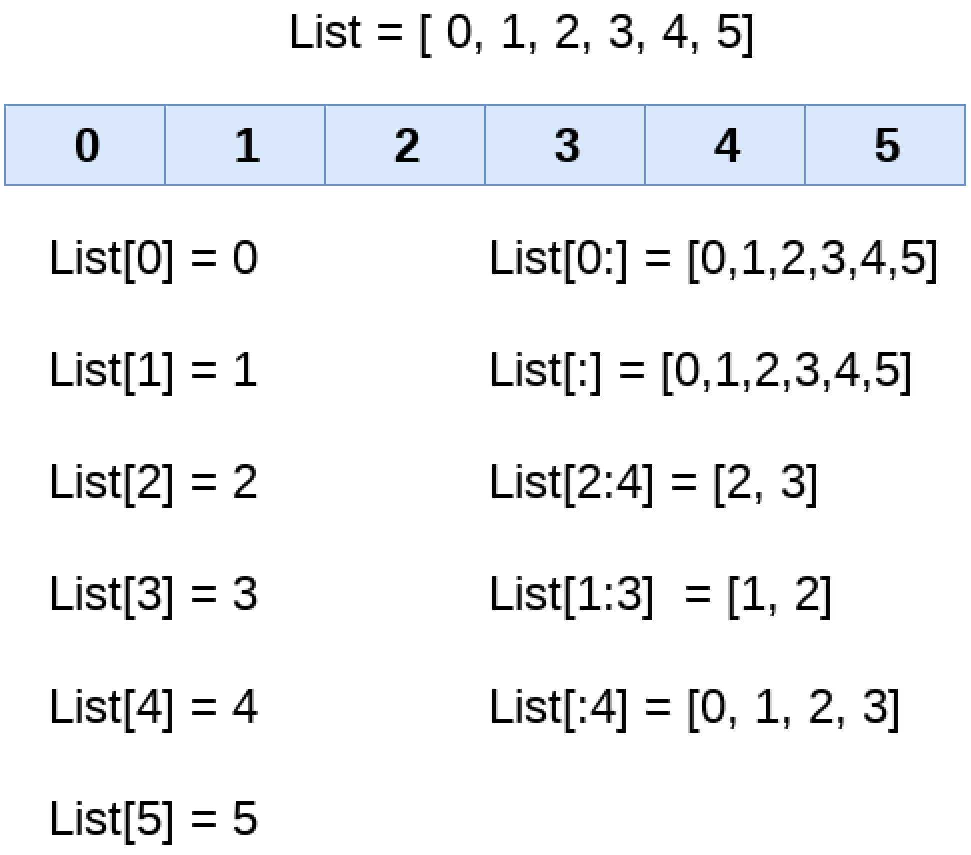
**Output:**

```
True
```

**List indexing and slicing:**

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



```
List = [ 0, 1, 2, 3, 4, 5]
```

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

```
List[0] = 0          List[0:] = [0,1,2,3,4,5]
List[1] = 1          List[:] = [0,1,2,3,4,5]
List[2] = 2          List[2:4] = [2, 3]
List[3] = 3          List[1:3]  = [1, 2]
List[4] = 4          List[:4] = [0, 1, 2, 3]
List[5] = 5
```

We can get the sub-list of the list using the following syntax.

**list_varible(start:stop:step)**

- The **start** denotes the starting index position of the list.
- The **stop** denotes the last index position of the list.
- The **step** is used to skip the nth element within a **start:stop**

Consider the following example:

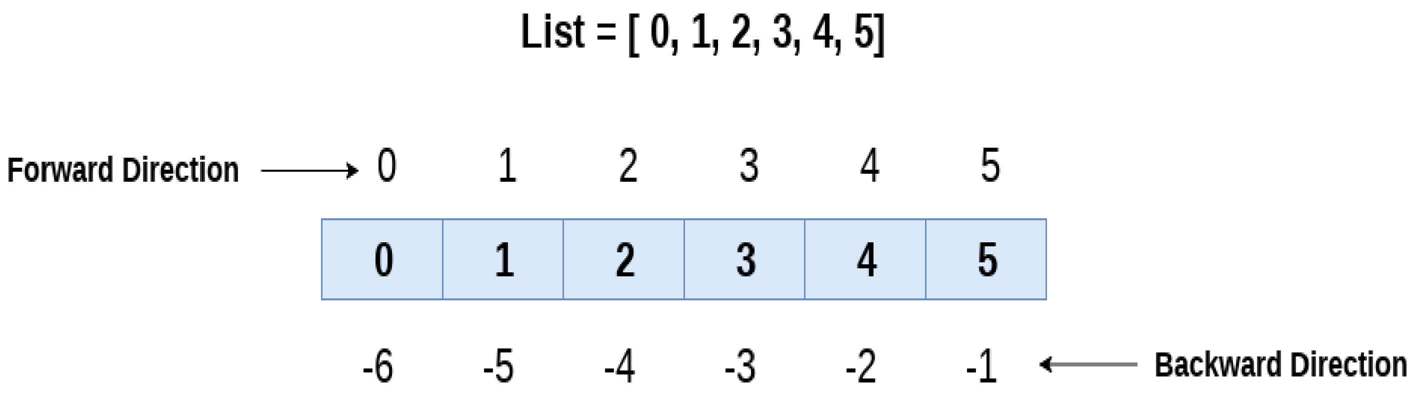
```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
```

```
# Slicing the elements
print(list[0:6])
# By default the index value is 0 so its starts from the 0th element and go for index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

**Output:**

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.



Let's have a look at the following example where we will use negative indexing to access the elements of the list.

```
list = [1,2,3,4,5]
print(list[-1])
print(list[-3:])
print(list[:-1])
print(list[-3:-1])
```

**Output:**

```
5
[3, 4, 5]
[1, 2, 3, 4]
[3, 4]
```

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

**Updating List values:**
Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the **slice** and **assignment** operator.
Python also provides **append()** and **insert()** methods, which can be used to add values to the list.

Consider the following example to update the values inside the list.

```
list = [1, 2, 3, 4, 5, 6]
print(list)
# It will assign value to the value to the second index
list[2] = 10
print(list)
# Adding multiple-element
list[1:3] = [89, 78]
print(list)
# It will add value at the end of the list
list[-1] = 25
print(list)
# It will append value at the end of the list
list.append(10)
print(list)
# It will insert value at index 2 in the list
list.insert(2,20)
print(list)
```

**Output:**

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
[1, 89, 78, 4, 5, 25,10]
[1, 89, 20,78, 4, 5, 25,10]
```

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove()** method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

```
list = [1, 2, 3, 4, 5, 6]
print(list)
```

```
del list[1:3]
print(list)
# It will delete the values from index 1 to index 2
list.remove(6)
print(list)
# It will delete the first occurrence of value 6 from the list and it will throws an error if element is not present in the list
```

**Output:**

```
[1, 2, 3, 4, 5, 6]
[1, 4, 5, 6]
[1, 4, 5]
```

**Python List Operations:**

The concatenation (+) and repetition (*) operators work in the same way as they were working with the strings.

Let's see how the list responds to various operators.

Consider a Lists l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8] to perform operation.

| **Operator** | **Description** | **Example** |
|---|---|---|
| Repetition | The repetition operator enables the list elements to be repeated multiple times. | l1*2 = [1, 2, 3, 4, 1, 2, 3, 4] |
| Concatenation | It concatenates the list mentioned on either side of the operator. | l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8] |
| Membership | It returns true if a particular item exists in a particular list otherwise false. | print(2 in l1) prints True. |
| Iteration | The for loop is used to iterate over the list elements. | for i in l1:<br>    print(i)<br>**Output**<br>1<br>2<br>3<br>4 |
| Length | It is used to get the length of the list | len(l1) = 4 |

**Iterating a List:**
A list can be iterated by using a **for - in** loop. A simple list containing four strings, which can be iterated as follows.

```
list = ["John", "David", "James", "Jonathan"]
for i in list:
# The i variable will iterate over the elements of the List and contains each element in each i
teration.
   print(i)
```

**Output:**

```
John
David
James
Jonathan
```

**Adding elements to the list:**
Python provides **append()** and **insert()** functions which are used to add an element to the list. Insert() function will insert the element to a particular index, however, the append() function can only add value to the end of the list.

Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

```
#Declaring the empty list
l =[]
#Number of elements will be entered by the user
n = int(input("Enter the number of elements in the list:"))
# for loop to take the input
for i in range(0,n):
   # The input is taken from the user and added to the list as the item
   l.append(input("Enter the item:"))
print("printing the list items..")
# traversal loop to print the list items
for i in l:
   print(i, end = "  ")

l.insert(2,50)
print("printing the list items after insertion..")
for i in l:
   print(i, end = "  ")
```

**Output:**

```
Enter the number of elements in the list:5
Enter the item:25
Enter the item:46
Enter the item:12
Enter the item:75
Enter the item:42
printing the list items..
25  46  12  75  42
printing the list items after insertion..
25  46  50  12  75  42
```

**Python List Built-in functions:**
Python provides the following built-in functions, which can be used with the lists.

| SN | Function | Description | Example |
|---|---|---|---|
| 1 | cmp(list1, list2) | It compares the elements of both the lists. | This method is not used in the Python 3 and the above versions. |
| 2 | len(list) | It is used to calculate the length of the list. | L1 = [1,2,3,4,5,6,7,8]<br>print(len(L1))<br>8 |
| 3 | max(list) | It returns the maximum element of the list. | L1 = [12,34,26,48,72]<br>print(max(L1))<br>72 |
| 4 | min(list) | It returns the minimum element of the list. | L1 = [12,34,26,48,72]<br>print(min(L1))<br>12 |
| 5 | list.sort() | It sorts the list ascending by default. | cars = ['Ford', 'BMW', 'Volvo']<br>print(cars.sort())<br>['Ford', 'BMW', 'Volvo'] |
| 5 | list(seq) | It converts any sequence to the list. | str = "Johnson"<br>s = list(str)<br>print(type(s))<br><class list> |

**Let's have a look at the few list examples:**

**Example: 1-** Write a program to find the sum of the element in the list.

```
list1 = [3,4,5,9,10,12,24]
sum = 0
for i in list1:
    sum = sum+i
print("The sum is:",sum)
```

**Output:**

```
The sum is: 67
```

**Example: 2-** Write the program to find the common elements from lists.

```
list1 = [1,2,3,4,5,6]
list2 = [7,8,9,2,10]
for x in list1:
    for y in list2:
        if x == y:
            print("The common element is:",x)
```

**Output:**

```
The common element is: 2
```

**Example: 3-** Write the program to remove the duplicate elements of the list.

```
list1 = [1,2,2,3,55,98,65,65,13,29]
# Declare an empty list that will store unique values
list2 = []
for i in list1:
    if i not in list2:
        list2.append(i)
print(list2)
```

**Output:**

```
[1, 2, 3, 55, 98, 65, 13, 29]
```

## Python Tuple:

Python Tuple is used to store the sequence of immutable Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

**Creating a tuple:**

- A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use. A tuple can be defined as follows.

```
T1 = (101, "Peter", 22)
T2 = ("Apple", "Banana", "Orange")
T3 = 10,20,30,40,50

print(type(T1))
print(type(T2))
print(type(T3))
```

**Output:**

```
<class 'tuple'>
<class 'tuple'>
<class 'tuple'>
```

*Note: The tuple which is created without using parentheses is also known as tuple packing.*

- An empty tuple can be created as follows.

```
T4 = ()
```

- Creating a tuple with single element is slightly different. We will need to put comma after the element to declare the tuple.

```
tup1 = ("JavaTpoint")
print(type(tup1))
#Creating a tuple with single element
tup2 = ("JavaTpoint",)
print(type(tup2))
```

**Output:**

```
<class 'str'>
<class 'tuple'>
```

**Tuple indexing and slicing:**

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following example of tuple:

**Example-1:**

```
tuple1 = (10, 20, 30, 40, 50)
print(tuple1)
for i in tuple1:
    print(i)
```

**Output:**

```
(10, 20, 30, 40, 50)
10
20
30
40
50
```

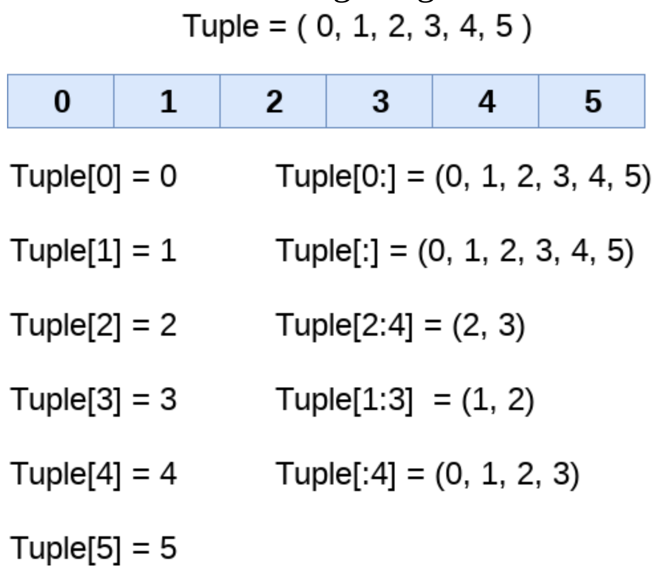
**Example-2:**

```
tuple1 = tuple(input("Enter the tuple elements ..."))
print(tuple1)
for i in tuple1:
    print(i)
```

**Output:**

```
Enter the tuple elements ...12345
('1', '2', '3', '4', '5')
1
2
3
4
5
```

Consider the following image to understand the indexing and slicing in detail.



```
Tuple = ( 0, 1, 2, 3, 4, 5 )

Tuple[0] = 0        Tuple[0:] = (0, 1, 2, 3, 4, 5)
Tuple[1] = 1        Tuple[:] = (0, 1, 2, 3, 4, 5)
Tuple[2] = 2        Tuple[2:4] = (2, 3)
Tuple[3] = 3        Tuple[1:3]  = (1, 2)
Tuple[4] = 4        Tuple[:4] = (0, 1, 2, 3)
Tuple[5] = 5
```

Consider the following example:

```
tup = (1,2,3,4,5,6,7)
print(tup[0])
print(tup[1])
print(tup[2])
# It will give the IndexError
print(tup[8])
```

**Output:**

```
1
2
3
tuple index out of range
```

In the above code, the tuple has 7 elements which denote 0 to 6. We tried to access an element outside of tuple that raised an **IndexError**.

```
tuple = (1,2,3,4,5,6,7)
#element 1 to end
print(tuple[1:])
#element 0 to 3 element
print(tuple[:4])
#element 1 to 4 element
print(tuple[1:5])
print(tuple[0:6:2])          # element 0 to 6 and take step of 2
```

**Output:**

```
(2, 3, 4, 5, 6, 7)
(1, 2, 3, 4)
(1, 2, 3, 4)
(1, 3, 5)
```

**Negative Indexing:**

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on.

The elements from left to right are traversed using the negative indexing. Consider the following example:

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
print(tuple1[-4])
print(tuple1[-3:-1])
print(tuple1[:-1])
print(tuple1[-2:])
```

**Output:**

```
5
2
(3, 4)
(1, 2, 3, 4)
(4, 5)
```

**Deleting Tuple:**

Unlike lists, the tuple items cannot be deleted by using the **del** keyword as tuples are immutable. To delete an entire tuple, we can use the **del** keyword with the tuple name.

Consider the following example.

```
tuple1 = (1, 2, 3, 4, 5, 6)
print(tuple1)
del tuple1[0]
print(tuple1)
del tuple1
print(tuple1)
```

**Output:**

```
(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "tuple.py", line 4, in <module>
    print(tuple1)
NameError: name 'tuple1' is not defined
```

**Basic Tuple Operations:**

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

| **Operator** | **Description** | **Example** |
|---|---|---|
| Repetition | The repetition operator enables the tuple elements to be repeated multiple times. | T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5) |
| Concatenation | It concatenates the tuple mentioned on either side of the operator. | T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9) |
| Membership | It returns true if a particular item exists in the tuple otherwise false | print (2 in T1) prints True. |
| Iteration | The for loop is used to iterate over the tuple elements. | for i in T1:<br>    print(i)<br>**Output** |

| | | |
|---|---|---|
| | | 1<br>2<br>3<br>4<br>5 |
| Length | It is used to get the length of the tuple. | len(T1) = 5 |

**Python Tuple inbuilt functions:**

| SN | Function | Description |
|---|---|---|
| 1 | len(tuple) | It calculates the length of the tuple. |
| 2 | max(tuple) | It returns the maximum element of the tuple |
| 3 | min(tuple) | It returns the minimum element of the tuple. |
| 4 | tuple(seq) | It converts the specified sequence to the tuple. |

**Where use tuple?**
Using tuple instead of list is used in the following scenario.
1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.
2. Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.

```
[(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]
```

**List vs. Tuple:**

| SN | List | Tuple |
|---|---|---|
| 1 | The literal syntax of list is shown by the []. | The literal syntax of the tuple is shown by the (). |
| 2 | The List is mutable. | The tuple is immutable. |
| 3 | The List has the variable length. | The tuple has the fixed length. |
| 4 | The list provides more functionality than a tuple. | The tuple provides less functionality than the list. |
| 5 | The list is used in the scenario in which we need to store the simple collections | The tuple is used in the cases where we need to store the read-only collections i.e., the |

| | | |
|---|---|---|
| | with no constraints where the value of the items can be changed. | value of the items cannot be changed. It can be used as the key inside the dictionary. |
| 6 | The lists are less memory efficient than a tuple. | The tuples are more memory efficient because of its immutability. |

**Can we change the items of a tuple?**

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

## Python Dictionary:

Python Dictionary is used to store the data in a **key-value pair** format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the **mutable data-structure**. The dictionary is defined into element Keys and values.

- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the **value can be any Python object**. In contrast, the **keys are the immutable Python object**, i.e., Numbers, string, or tuple.

**Creating the dictionary:**

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:). The syntax to define the dictionary is given below.

**Syntax: {Key:Value}**

```
Dict = {"Name": "Tom", "Age": 22}
```

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

**Example:** Let's see an example to create a dictionary and print its content.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
```

**Output:**

```
<class 'dict'>
Printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary with dict() method
```

```
Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
print("\nCreate Dictionary by using  dict(): ")
print(Dict)

# Creating a Dictionary with each item as a Pair
Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

**Output:**

```
Empty Dictionary:
{}

Create Dictionary by using dict():
{1: 'Java', 2: 'T', 3: 'Point'}

Dictionary with each item as a pair:
{1: 'Devansh', 2: 'Sharma'}
```

**Accessing the dictionary values:**

We have discussed how the data can be accessed in the list and tuple by using the indexing. However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print("Name :”, Employee["Name"])
print("Age :”, Employee["Age"])
print("Salary :”, Employee["salary"])
print("Company :”, Employee["Company"])
```

**Output:**

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Python provides us with an alternative to use the **get()** method to access the dictionary values. It would give the same result as given by the indexing.

**Adding dictionary values:**
The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The **update()** method is also used to update an existing value.

**Note:** If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.
Let's see an example to update the dictionary values.

**Example-1:**

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Adding elements to dictionary one at a time
Dict[0] = 'Peter'
Dict[2] = 'Joseph'
Dict[3] = 'Ricky'
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# with a single Key
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[3] = 'JavaTpoint'
print("\nUpdated key value: ")
print(Dict)
```

**Output:**

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}
```

```
Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

**Example-2:**

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....");
Employee["Name"] = input("Name: ");
Employee["Age"] = int(input("Age: "));
Employee["salary"] = int(input("Salary: "));
Employee["Company"] = input("Company:");
print("printing the new data");
print(Employee)
```

**Output:**

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

**Deleting elements using del keyword:**
The items of the dictionary can be deleted by using the **del** keyword as given below.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee");
del Employee
print("Lets try to print it again ");
print(Employee)
```

**Output:**

```
<class 'dict'>
printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

**Using pop() method:** The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.

```
# Creating a Dictionary
Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
# Deleting a key
# using pop() method
pop_ele = Dict.pop(3)
print(Dict)
```

**Output:**

```
{1: 'JavaTpoint', 2: 'Peter'}
```

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

**Iterating Dictionary**

A dictionary can be iterated using for loop as given below.

**Example-1:**

```
# for loop to print all the keys of a dictionary

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
   print(x)
```

**Output:**

```
Name
Age
salary
Company
```

**Example-2:**
**#for loop to print all the values of the dictionary**

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
   print(Employee[x])
```

**Output:**

```
John
29
25000
GOOGLE
```

**Example–3:**
**#for loop to print the values of the dictionary by using values() method.**

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.values():
   print(x)
```

**Output:**

```
John
29
25000
GOOGLE
```

**Example-4:**
**#for loop to print the items of the dictionary by using items() method.**

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.items():
   print(x)
```

**Output:**

```
('Name', 'John')
('Age', 29)
('salary', 25000)
('Company', 'GOOGLE')
```

**Properties of Dictionary keys:**

**1.** In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.
Consider the following example.

```
Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}
for x,y in Employee.items():
    print(x,y)
```

**Output:**

```
Name John
Age 29
Salary 25000
Company GOOGLE
```

**2.** In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.
Consider the following example.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
for x,y in Employee.items():
    print(x,y)
```

**Output:**

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in
    Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

**Built-in Dictionary functions:**

The built-in python dictionary methods along with the description are given below.

| SN | Function | Description |
|---|---|---|
| 1 | len(dict) | It is used to calculate the length of the dictionary. |
| 2 | type(variable) | It is used to print the type of the passed variable. |
| 3 | dict.clear() | It is used to delete all the items of the dictionary. |
| 4 | dict.copy() | It returns a shallow copy of the dictionary. |

| | | |
|---|---|---|
| 5 | dict.items() | It returns all the key-value pairs as a tuple. |
| 6 | dict.keys() | It returns all the keys of the dictionary. |
| 7 | dict.update(dict2) | It updates the dictionary by adding the key-value pair of dict2 to this dictionary. |
| 8 | dict.values() | It returns all the values of the dictionary. |

## Python Function:

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

Python provide us various inbuilt functions like **range()** or **print()**. Although, the user can create its functions, which can be called user-defined functions.

**There are mainly two types of functions:**

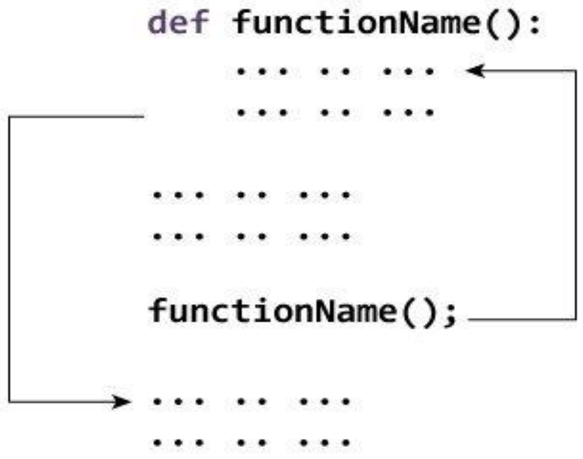
- **User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.
- **Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

**Advantage of Functions in Python:**

There are the following advantages of Python functions.

- Using functions, we can avoid rewriting the same logic/code again and again in a program.
- We can call Python functions multiple times in a program and anywhere in a program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.

**How Function works in Python?**



```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName();

... .. ...
... .. ...
```

**Creating a Function:**
Python provides the **def** keyword to define the function. The syntax of the define function is given below.

**Syntax:**

```
def my_function(parameters):
      function_block
      return expression
```

Let's understand the syntax of functions definition.

- The **def** keyword, along with the function name is used to define the function.
- The identifier rule must follow the function name.
- A function accepts the parameter (argument), and they can be optional.
- The function block is started with the colon (:), and block statements must be at the same indentation.
- The **return** statement is used to return the value. A function can have only one **return**

**Function Calling:**
In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

Consider the following example of a simple example that prints the message "Hello World".

```
#function definition
def hello_world():
    print("hello world")
# function calling
hello_world()
```

**Output:**

```
  hello world
```

**The return statement:**
The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

**Syntax:**

```
return [expression_list]
```

It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

**Consider the following examples:**

**Example-1:**

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is:",sum())
```

**Output:**

```
  The sum is: 30
```

In the above code, we have defined the function named **sum,** and it has a statement **c = a+b,** which computes the given values, and the result is returned by the return statement to the caller function.

**Example-2:** Creating function without return statement

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
# calling sum() function in print statement
print(sum())
```

**Output:**

```
None
```

In the above code, we have defined the same function without the return statement as we can see that the **sum()** function returned the **None** object to the caller function.

**Example-3:** Program for absolute value.

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num
```

```
print(absolute_value(2))
print(absolute_value(-4))
```

**Output:**

```
2
4
```

**Example-4:** Print the even numbers from a given list

```
def is_even_num(l):
    enum = []
    for n in l:
        if n % 2 == 0:
            enum.append(n)
    return enum

e= is_even_num([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(e)
```

**Output:**

```
[2, 4, 6, 8]
```

**Example-5:** Find the factorial of a given number.

```
def fact(n):
    f=1
    for i in range(1,n+1):
        f=f*i
    return f
print(fact(10))
```

**Arguments in function:**

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

Consider the following example, which contains a function that accepts a string as the argument.

**Example-1:**

```
#defining the function
def func (name):
    print("Hi ",name)
#calling the function
func("Devansh")
```

**Output:**

```
Hi Devansh
```

**Example-2:**

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

**Output:**

```
Enter a: 10
Enter b: 20
Sum =  30
```

**Types of arguments:**

There may be several types of arguments which can be passed at the time of function call.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

**Required Arguments:**

Till now, we have learned about function calling in Python. However, we can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.
Consider the following example.

**Example-1:**

```
def func(name):
    message = "Hi "+name
    return message
name = input("Enter the name:")
print(func(name))
```

**Output:**
```
Enter the name: John
Hi John
```

**Example-2:**
```
#the function simple_interest accepts three arguments and returns the simple interest accord
ingly
def simple_interest(p,t,r):
   return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

**Output:**
```
Enter the principle amount: 5000
Enter the rate of interest: 5
Enter the time in years: 3
Simple Interest:  750.0
```

**Example-3:**
```
#the function calculate returns the sum of two arguments a and b
def calculate(a,b):
   return a+b
calculate(10) # this causes an error as we are missing a required arguments b.
```

**Output:**
```
TypeError: calculate() missing 1 required positional argument: 'b'
```

**Default Arguments:**
Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

**Example-1:**
```
def printme(name,age=22):
   print("My name is",name,"and age is",age)
printme(name = "john")
```

**Output:**
```
My name is John and age is 22
```

**Example-2:**

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function
printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age
```

**Output:**

```
  My name is john and age is 22
  My name is David and age is 10
```

**Scope of variables:**
The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.
In python, the variables are defined with the two types of scopes.

1. Local variables
2. Global variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.

**Local Scope:**
A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

**Example:** A variable created inside a function is available inside that function.

```
def myfunc():
  x = 300
  print(x)
myfunc()
```

**Global Scope:**
A variable created in the main body of the Python code is a global variable and belongs to the global scope. Global variables are available from within any scope, global and local.

**Example:** A variable created outside of a function is global and can be used by anyone

```
x = 300
def myfunc():
  print(x)
myfunc()
print(x)
```

## Differences between Break, Continue and Pass statements:

**Break Statement in Python:**
The break statement in Python is used to terminate the loop or statement in which it is present. After that, the control will pass to the statements that are present after the break statement, if available. If the break statement is present in the nested loop, then it terminates only inner loops which contain the break statement.

**Syntax of Break Statement:**
The break statement in Python has the following syntax:

```
for / while loop:
    # statement(s)
    if condition:
        break
    # statement(s)
# loop end
```

**Example-1:**

```
s = 'geeksforgeeks'
# Using for loop
for letter in s:
    print(letter)
    # break the loop as soon it sees 'e' or 's'
    if letter == 'e' or letter == 's':
        break
print("Out of for loop")
print()
```

**Output:**

```
g
e
Out of for loop
```

**Example-2:**

```
# first for loop
for i in range(1, 5):
     # second for loop
    for j in range(2, 6):
         # break the loop if
        # j is divisible by i
        if j%i == 0:
            break
        print(i, " ", j)
```

**Output:**

```
3   2
4   2
4   3
```

**Continue Statement in Python:**

Continue is also a loop control statement just like the break statement. continue statement is opposite to that of the break statement, instead of terminating the loop, it forces to execute the next iteration of the loop. As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

**Syntax of Continue Statement:**

The continue statement in Python has the following syntax:

```
for / while loop:
    # statement(s)
    if condition:
        continue
    # statement(s)
```

**Example:**

```
# loop from 1 to 10
for i in range(1, 11):

    # If i is equals to 6,
    # continue to next iteration
    # without printing
    if i == 6:
        continue
    else:
        # otherwise print the value
        # of i
        print(i, end = " ")
```

**Output:**

```
1 2 3 4 5 7 8 9 10
```

**Pass Statement in Python:**
As the name suggests pass statement simply does nothing. The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. It is like a null operation, as nothing will happen if it is executed. Pass statements can also be used for writing empty loops. Pass is also used for empty control statements, functions, and classes.

**Syntax of Pass Statement:**
The pass statement in Python has the following syntax:
```
function/ condition / loop:
   pass
```

**Example:**
```
# Pass statement
s = "geeks"
for i in s:
    if i == 'k':
        print('Pass executed')
        pass
    print(i)
```

**Output:**
```
g
e
e
Pass executed
k
s
```

## Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.
Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

### Creating Classes in Python:

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

**Syntax**

```
class ClassName:
   #statement_suite
```

## Objects in Python:

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python.

**Syntax:**

```
# Declare an object of a class
object_name = Class_Name(arguments)
```

## Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

**Syntax**

```
class derived-class(base class):
   <class-suite>
```

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

**Syntax**
**class** derive-**class**(<base **class** 1>, <base **class** 2>, ..... <base **class** n>):
    <**class** - suite>

## Python RegEx:

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern.

**RegEx Module:**
Python has a built-in package called re, which can be used to work with Regular Expressions.

**Syntax to import the re module:**
```
import re
```
When you have imported the re module, you can start using regular expressions:

**Example:** Search the string to see if it starts with "The" and ends with "Spain"

```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
if x:
  print("YES! We have a match!")
else:
  print("No match")
```
**Output**
```
YES! We have a match!
```

**RegEx Functions:** The re module offers a set of functions that allows us to search a string for a match.

| Function | Description |
|---|---|
| findall | Returns a list containing all matches |
| search | Returns a Match object if there is a match anywhere in the string |
| split | Returns a list where the string has been split at each match |
| sub | Replaces one or many matches with a string |

**Metacharacters:** Metacharacters are characters with a special meaning:

| Character | Description | Example |
|---|---|---|
| [] | A set of characters | "[a-m]" |

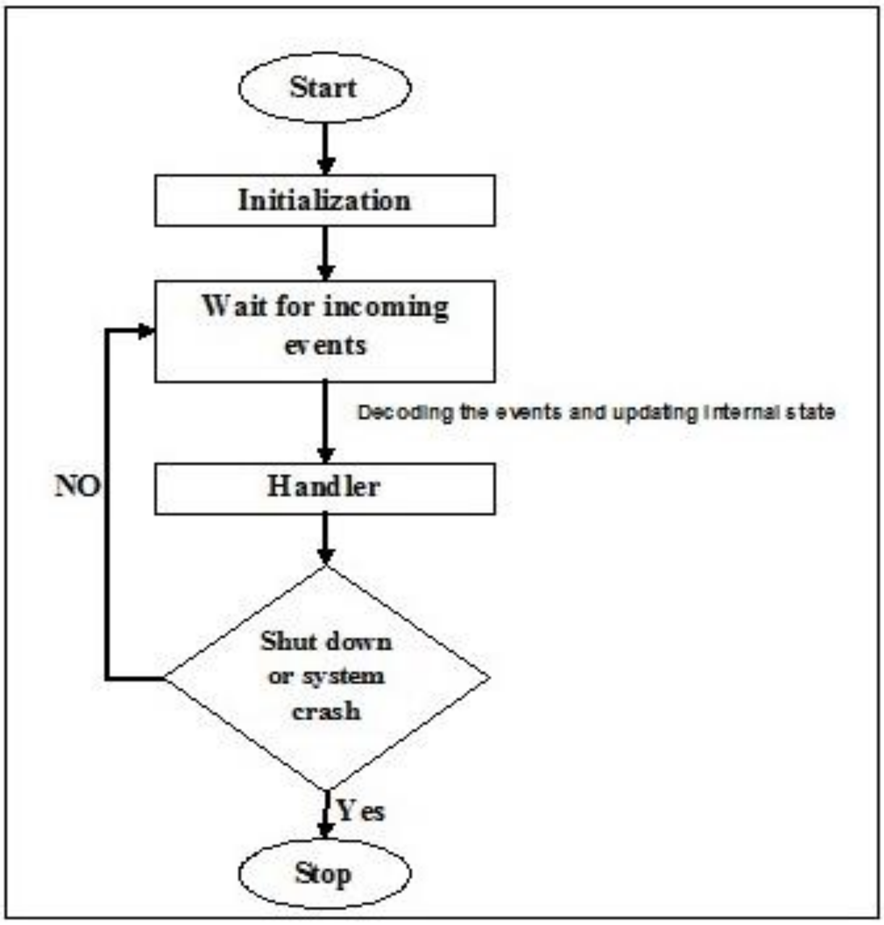
| | | |
|---|---|---|
| \ | Signals a special sequence (can also be used to escape special characters) | "\d" |
| . | Any character (except newline character) | "he..o" |
| ^ | Starts with | "^hello" |
| $ | Ends with | "planet$" |
| * | Zero or more occurrences | "he.*o" |
| + | One or more occurrences | "he.+o" |
| ? | Zero or one occurrences | "he.?o" |
| {} | Exactly the specified number of occurrences | "he.{2}o" |
| \| | Either or | "falls\|stays" |
| () | Capture and group | |

## Event-Driven Programming:

Eventually, the flow of a program depends upon the events, and programming which focuses on events is called Event-Driven programming. Generally the flow of events depends on either parallel or sequential models, but in Event-Driven Programming it will also depends on the asynchronous model. The programming model following the concept of Event-Driven programming is called the Asynchronous model. The working of Event-Driven programming depends upon the events happening in a program.

Other than this, it depends upon the program's event loops that always listen to a new incoming event in the program. Once an event loop starts in the program, then only the events will decide what will execute and in which order.

Look at the following flow chart of event loops to understand the working of events in event-driven programming:


Start
Initialization
Wait for incoming events
Decoding the events and updating internal state
Handler
NO
Shut down or system crash
Yes
Stop

## GUI Programming:

Python provides various options for developing graphical user interfaces (GUIs). The most important methods are listed below.

- **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python.
- **wxPython** − This is an open-source Python interface for wxWidgets GUI toolkit.

- **PyQt** − This is also a Python interface for a popular cross-platform Qt GUI library.
- **PyGTK** − PyGTK is a set of wrappers written in Python and C for GTK + GUI library.
- **PySimpleGUI** − PySimpleGui is an open source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide and WxPython toolkits.
- **Pygame** − Pygame is a popular Python library used for developing video games. It is free, open source and cross-platform wrapper around Simple DirectMedia Library (SDL).
- **Jython** − Jython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machine.

## Packages and Modules:

In Python, both modules and packages organize and structure the code but serve different purposes. In simple terms, a module is a single file containing python code, whereas a package is a collection of modules that are organized in a directory hierarchy.

**Module in Python:**
In Python, a module is a single file containing Python definitions and statements. These definitions and statements can include variables, functions, and classes and can be used to organize related functionality into a single, reusable package. Module organizes and reuses code in Python by grouping related code into a single file.
Modules can be imported and used in other Python files using the import statement.
Some popular modules in Python are math, random, csv, and datetime.

**Example:** Consider a Python module math.py that contains a function to calculate the square of a number

```
#math.py module
def square(i):
    return x**2
```

This module can be used be imported and used in the different files as follows:

```
#main.py file
import math
print(math.square(5)) #output 25
```

**Package in Python:**
Python Packages are collections of modules that provide a set of related functionalities, and these modules are organized in a directory hierarchy. In simple terms, packages in Python are a way of organizing related modules in a single namespace.

- Packages in Python are installed using a package manager like pip (a tool for installing and managing Python packages).
- Each Python package must contain a file named _init_.py.

**Example:** Let there be any package (named my_package) that contains two sub-modules (mod_1, and mod_2)

```
my_package/
  _init_.py
  mod_1.py
  mod_2.py
```

**Difference between Module and Package in Python:**

| **Parameter** | **Module** | **Package** |
|---|---|---|
| **Definition** | It can be a simple Python file (.py extensions) that contains collections of functions and global variables. | A Package is a collection of different modules with an _init_.py file. |
| **Purpose** | Code organization | Code distribution and reuse |
| **Organization** | Code within a single file | Related modules in a directory hierarchy |
| **Sub-modules** | None | Multiple sub-modules and sub-packages |
| **Required Files** | Only Python File(.py format) | '_init_.py' file and python files |
| **How to import** | import module_name | import package_name.module_name |
| **Example** | math, random, os, datetime, csv | Numpy, Pandas, Matplotlib, django |